Cloud Tools Overview
Hadoop
Outline

• Hadoop - Basics
• HDFS
  – Goals
  – Architecture
  – Other functions
• MapReduce
  – Basics
  – Word Count Example
  – Handy tools
  – Finding shortest path example
• Related Apache sub-projects (Pig, HBase, Hive)
Hadoop - Why?

• Need to process huge datasets on large clusters of computers
• Very expensive to build reliability into each application
• Nodes fail every day
  – Failure is expected, rather than exceptional
  – The number of nodes in a cluster is not constant
• Need a common infrastructure
  – Efficient, reliable, easy to use
  – Open Source, Apache Licence
Who uses Hadoop?

- Amazon/A9
- Facebook
- Google
- New York Times
- Veoh
- Yahoo!
- .... many more
Commodity Hardware

- Typically in 2 level architecture
  - Nodes are commodity PCs
  - 30-40 nodes/rack
  - Uplink from rack is 3-4 gigabit
  - Rack-internal is 1 gigabit
Hadoop Distributed File System (HDFS)

Original Slides by
Dhruba Borthakur
Apache Hadoop Project Management Committee
Goals of HDFS

• Very Large Distributed File System
  – 10K nodes, 100 million files, 10PB

• Assumes Commodity Hardware
  – Files are replicated to handle hardware failure
  – Detect failures and recover from them

• Optimized for Batch Processing
  – Data locations exposed so that computations can move to where data resides
  – Provides very high aggregate bandwidth
Distributed File System

- Single Namespace for entire cluster
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Files are broken up into blocks
  - Typically 64MB block size
  - Each block replicated on multiple DataNodes
- Intelligent Client
  - Client can find location of blocks
  - Client accesses data directly from DataNode
HDFS Architecture

Metadata (Name, replicas, ...): /home/foo/data, 3, ...

Metadata ops

Block ops

Replication

Rack 1

Rack 2
Functions of a NameNode

• Manages File System Namespace
  – Maps a file name to a set of blocks
  – Maps a block to the DataNodes where it resides

• Cluster Configuration Management

• Replication Engine for Blocks
NameNode Metadata

• Metadata in Memory
  – The entire metadata is in main memory
  – No demand paging of metadata

• Types of metadata
  – List of files
  – List of Blocks for each file
  – List of DataNodes for each block
  – File attributes, e.g. creation time, replication factor

• A Transaction Log
  – Records file creations, file deletions etc
DataNode

• A Block Server
  – Stores data in the local file system (e.g. ext3)
  – Stores metadata of a block (e.g. CRC)
  – Serves data and metadata to Clients

• Block Report
  – Periodically sends a report of all existing blocks to the NameNode

• Facilitates Pipelining of Data
  – Forwards data to other specified DataNodes
Block Placement

• Current Strategy
  – One replica on local node
  – Second replica on a remote rack
  – Third replica on same remote rack
  – Additional replicas are randomly placed

• Clients read from nearest replicas

• Would like to make this policy pluggable
Heartbeats

• DataNodes send heartbeat to the NameNode
  – Once every 3 seconds
• NameNode uses heartbeats to detect DataNode failure
Replication Engine

- NameNode detects DataNode failures
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes
Data Correctness

• Use Checksums to validate data
  – Use CRC32

• File Creation
  – Client computes checksum per 512 bytes
  – DataNode stores the checksum

• File access
  – Client retrieves the data and checksum from DataNode
  – If Validation fails, Client tries other replicas
NameNode Failure

- A single point of failure
- Transaction Log stored in multiple directories
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)
- Need to develop a real HA solution
Data Pieplining

• Client retrieves a list of DataNodes on which to place replicas of a block
• Client writes block to the first DataNode
• The first DataNode forwards the data to the next node in the Pipeline
• When all replicas are written, the Client moves on to write the next block in file
Rebalancer

• **Goal:** % disk full on DataNodes should be similar
  – Usually run when new DataNodes are added
  – Cluster is online when Rebalancer is active
  – Rebalancer is throttled to avoid network congestion
  – Command line tool
Secondary NameNode

- Copies FsImage and Transaction Log from Namenode to a temporary directory
- Merges FSImage and Transaction Log into a new FSImage in temporary directory
- Uploads new FSImage to the NameNode
  - Transaction Log on NameNode is purged
User Interface

• Commads for HDFS User:
  – hadoop dfs -mkdir /foodir
  – hadoop dfs -cat /foodir/myfile.txt
  – hadoop dfs -rm /foodir/myfile.txt

• Commands for HDFS Administrator
  – hadoop dfsadmin -report
  – hadoop dfsadmin -decommision datanodename

• Web Interface
  – http://host:port/dfshealth.jsp
MapReduce

Original Slides by
Owen O’Malley (Yahoo!)
&
Christophe Bisciglia, Aaron Kimball & Sierra Michells-Slettvet
MapReduce - What?

- MapReduce is a programming model for efficient distributed computing
- It works like a Unix pipeline
  - `cat input | grep | sort | uniq -c | cat > output`
  - **Input** | **Map** | **Shuffle & Sort** | **Reduce** | **Output**
- Efficiency from
  - Streaming through data, reducing seeks
  - Pipelining
- A good fit for a lot of applications
  - Log processing
  - Web index building
MapReduce - Dataflow

Pre-loaded local input data

Intermediate data from mappers

Values exchanged by shuffle process

Reducing process generates outputs

Outputs stored locally

Node 1
- Mapping process

Node 2
- Mapping process

Node 3
- Mapping process

Node 1
- Reducing process

Node 2
- Reducing process

Node 3
- Reducing process
MapReduce - Features

• Fine grained Map and Reduce tasks
  – Improved load balancing
  – Faster recovery from failed tasks

• Automatic re-execution on failure
  – In a large cluster, some nodes are always slow or flaky
  – Framework re-executes failed tasks

• Locality optimizations
  – With large data, bandwidth to data is a problem
  – Map-Reduce + HDFS is a very effective solution
  – Map-Reduce queries HDFS for locations of input data
  – Map tasks are scheduled close to the inputs when possible
Word Count Example

• Mapper
  – Input: value: lines of text of input
  – Output: key: word, value: 1

• Reducer
  – Input: key: word, value: set of counts
  – Output: key: word, value: sum

• Launching program
  – Defines this job
  – Submits job to cluster
The overall MapReduce word count process.
public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
    private static final IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public static void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasNext()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}
public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
    public static void map(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
Word Count Example

• Jobs are controlled by configuring *JobConfs*
• JobConfs are maps from attribute names to string values
• The framework defines attributes to control how the job is executed
  – `conf.set("mapred.job.name", "MyApp");`
• Applications can add arbitrary values to the JobConf
  – `conf.set("my.string", "foo");`
  – `conf.set("my.integer", 12);`
• JobConf is available to all tasks
Putting it all together

• Create a launching program for your application
• The launching program configures:
  – The *Mapper* and *Reducer* to use
  – The output key and value types (input types are inferred from the *InputFormat*)
  – The locations for your input and output
• The launching program then submits the job and typically waits for it to complete
Putting it all together

```java
JobConf conf = new JobConf(WordCount.class);
conf.setJobName("wordcount");

conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(Map.class);
conf.setCombinerClass(Reduce.class);
conf.setReducer(Reduce.class);

conf.setInputFormat(TextInputFormat.class);
Conf.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));

JobClient.runJob(conf);
```
Input and Output Formats

• A Map/Reduce may specify how its input is to be read by specifying an InputFormat to be used
• A Map/Reduce may specify how its output is to be written by specifying an OutputFormat to be used
• These default to TextInputFormat and TextOutputFormat, which process line-based text data
• Another common choice is SequenceFileInputFormat and SequenceFileOutputFormat for binary data
• These are file-based, but they are not required to be
How many Maps and Reduces

• Maps
  – Usually as many as the number of HDFS blocks being processed, this is the default
  – Else the number of maps can be specified as a hint
  – The number of maps can also be controlled by specifying the minimum split size
  – The actual sizes of the map inputs are computed by:
    • $\max(\min(\text{block\_size}, \text{data}/\#\text{maps}), \text{min\_split\_size})$

• Reduces
  – Unless the amount of data being processed is small
    • $0.95*\text{num\_nodes}*\text{mapred.tasktracker.tasks.maximum}$
Some handy tools

- Partitioners
- Combiners
- Compression
- Counters
- Speculation
- Zero Reduces
- Distributed File Cache
- Tool
Partitioners

- Partitioners are application code that define how keys are assigned to reduces
- Default partitioning spreads keys evenly, but randomly
  - Uses `key.hashCode() % num_reduces`
- Custom partitioning is often required, for example, to produce a total order in the output
  - Should implement `Partitioner` interface
  - Set by calling `conf.setPartitionerClass(MyPart.class)`
  - To get a total order, sample the map output keys and pick values to divide the keys into roughly equal buckets and use that in your partitioner
Combiners

- When *maps* produce many repeated keys
  - It is often useful to do a local aggregation following the *map*
  - Done by specifying a *Combiner*
  - Goal is to decrease size of the transient data
  - Combiners have the same interface as Reduces, and often are the same class
  - Combiners must **not** side effects, because they run an intermediate number of times
  - In *WordCount*, `conf.setCombinerClass(Reduce.class);`
Compression

- Compressing the outputs and intermediate data will often yield huge performance gains
  - Can be specified via a configuration file or set programmatically
  - Set `mapred.output.compress` to `true` to compress job output
  - Set `mapred.compress.map.output` to `true` to compress map outputs

- Compression Types (`mapred(.map)?.output.compression.type`)
  - “block” - Group of keys and values are compressed together
  - “record” - Each value is compressed individually
  - Block compression is almost always best

- Compression Codecs (`mapred(.map)?.output.compression.codec`)
  - Default (zlib) - slower, but more compression
  - LZO - faster, but less compression
Counters

• Often Map/Reduce applications have countable events
• For example, framework counts records in to and out of Mapper and Reducer
• To define user counters:
  static enum Counter {EVENT1, EVENT2};
  reporter.incrCounter(Counter.EVENT1, 1);
• Define nice names in a MyClass_Counter.properties file
  CounterGroupName=MyCounters
  EVENT1.name=Event 1
  EVENT2.name=Event 2
Speculative execution

• The framework can run multiple instances of slow tasks
  – Output from instance that finishes first is used
  – Controlled by the configuration variable `mapred.speculative.execution`
  – Can dramatically bring in long tails on jobs
Zero Reduces

• Frequently, we only need to run a filter on the input data
  – No sorting or shuffling required by the job
  – Set the number of reduces to 0
  – Output from maps will go directly to OutputFormat and disk
Distributed File Cache

- Sometimes need read-only copies of data on the local computer
  - Downloading 1GB of data for each Mapper is expensive
- Define list of files you need to download in JobConf
- Files are downloaded once per computer
- Add to launching program:
  ```java
  DistributedCache.addCacheFile(new URI("hdfs://nn:8020/foo"), conf);
  ```
- Add to task:
  ```java
  Path[] files = DistributedCache.getLocalCacheFiles(conf);
  ```
Tool

- Handle “standard” Hadoop command line options
  - -conf file - load a configuration file named file
  - -D prop=value - define a single configuration property prop
- Class looks like:

  public class MyApp extends Configured implements Tool {
    public static void main(String[] args) throws Exception {
      System.exit(ToolRunner.run(new Configuration(),
        new MyApp(), args));
    }
    public int run(String[] args) throws Exception {
      .... getConf() ....
    }
  }
Finding the Shortest Path

• A common graph search application is finding the shortest path from a start node to one or more target nodes
• Commonly done on a single machine with Dijkstra’s Algorithm
• Can we use BFS to find the shortest path via MapReduce?
Finding the Shortest Path: Intuition

• We can define the solution to this problem inductively
  – DistanceTo(startNode) = 0
  – For all nodes \( n \) directly reachable from startNode, \( \text{DistanceTo}(n) = 1 \)
  – For all nodes \( n \) reachable from some other set of nodes \( S \),
    \( \text{DistanceTo}(n) = 1 + \min(\text{DistanceTo}(m), \ m \in S) \)
From Intuition to Algorithm

• A map task receives a node $n$ as a key, and $(D, points-to)$ as its value
  – $D$ is the distance to the node from the start
  – $points-to$ is a list of nodes reachable from $n$

  $\forall p \in points-to$, emit $(p, D+1)$

• Reduces task gathers possible distances to a given $p$ and selects the minimum one
What This Gives Us

• This MapReduce task can advance the known frontier by one hop
• To perform the whole BFS, a non-MapReduce component then feeds the output of this step back into the MapReduce task for another iteration
  – Problem: Where’d the points-to list go?
  – Solution: Mapper emits \((n, \text{points-to})\) as well
• This algorithm starts from one node
• Subsequent iterations include many more nodes of the graph as the frontier advances
• Does this ever terminate?
  – Yes! Eventually, routes between nodes will stop being discovered and no better distances will be found. When distance is the same, we stop
  – Mapper should emit \((n,D)\) to ensure that “current distance” is carried into the reducer
Hadoop Subprojects
Hadoop Related Subprojects

• Pig
  – High-level language for data analysis
• HBase
  – Table storage for semi-structured data
• Zookeeper
  – Coordinating distributed applications
• Hive
  – SQL-like Query language and Metastore
• Mahout
  – Machine learning
Pig

Original Slides by
Matei Zaharia
UC Berkeley RAD Lab
Pig

- Started at Yahoo! Research
- Now runs about 30% of Yahoo!’s jobs
- Features
  - Expresses sequences of MapReduce jobs
  - Data model: nested “bags” of items
  - Provides relational (SQL) operators (JOIN, GROUP BY, etc.)
  - Easy to plug in Java functions
An Example Problem

- Suppose you have user data in a file, website data in another, and you need to find the top 5 most visited pages by users aged 18-25.
In Pig Latin

Users = \texttt{load} \texttt{`users'} \texttt{as} (name, age);
Filtered = \texttt{filter} Users \texttt{by} age \geq 18 \text{ and } age \leq 25;
Pages = \texttt{load} \texttt{`pages'} \texttt{as} (user, url);
Joined = \texttt{join} Filtered \texttt{by} name, Pages \texttt{by} user;
Grouped = \texttt{group} Joined \texttt{by} url;
Summed = \texttt{foreach} Grouped\texttt{generate} group,
  \hspace{1em} \texttt{count} (Joined) \text{ as } \texttt{clicks};
Sorted = \texttt{order} Summed \texttt{by} clicks \texttt{desc};
Top5 = \texttt{limit} Sorted 5;
\texttt{store} Top5 \texttt{into} \texttt{`top5sites'};
Ease of Translation

Load Users ➔ Filter by age ➔ Join on name ➔ Group on url ➔ Count clicks ➔ Order by clicks ➔ Take top 5

Load Pages ➔ Users = load ...

Fltrd = filter ...

Pages = load ...

Joined = join ...

Grouped = group ...

Summed = ... count()...

Sorted = order ...

Top5 = limit ...
Ease of Translation

Job 1
- Load Users
- Filter by age
- Join on name

Job 2
- Group on url
- Count clicks

Job 3
- Order by clicks
- Take top 5

Users = load ...
Fltrd = filter ...
Pages = load ...
Joined = join ...
Grouped = group ...
Summed = ... count()...
Sorted = order ...
Top5 = limit ...
HBase

Original Slides by
Tom White
Lexeme Ltd.
HBase - What?

• Modeled on Google’s Bigtable
• Row/column store
• Billions of rows/millions on columns
• Column-oriented - nulls are free
• Untyped - stores byte[]
### HBase - Data Model

<table>
<thead>
<tr>
<th>Row</th>
<th>Timestamp</th>
<th>Column family:</th>
<th>Column family:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>animal:</td>
<td>animal:</td>
</tr>
<tr>
<td>enclosure1</td>
<td></td>
<td>type</td>
<td>size</td>
</tr>
<tr>
<td>t2</td>
<td></td>
<td>zebra</td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td></td>
<td>lion</td>
<td>big</td>
</tr>
<tr>
<td>enclosure2</td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>repairs:</td>
<td>cost</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000 EUR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
**HBase - Data Storage**

Column family animal:

<table>
<thead>
<tr>
<th>(enclosure1, t2, animal:type)</th>
<th>zebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>(enclosure1, t1, animal:size)</td>
<td>big</td>
</tr>
<tr>
<td>(enclosure1, t1, animal:type)</td>
<td>lion</td>
</tr>
</tbody>
</table>

Column family repairs:

| (enclosure1, t1, repairs:cost) | 1000 EUR    |
HBase - Code

HTable table = ...
Text row = new Text("enclosure1");
Text col1 = new Text("animal:type");
Text col2 = new Text("animal:size");
BatchUpdate update = new BatchUpdate(row);
update.put(col1, "lion".getBytes("UTF-8"));
update.put(col2, "big".getBytes("UTF-8"));
table.commit(update);

update = new BatchUpdate(row);
update.put(col1, "zebra".getBytes("UTF-8"));
table.commit(update);
HBase - Querying

- Retrieve a cell
  Cell = table.getRow("enclosure1").getColumn("animal:type").getValue();

- Retrieve a row
  RowResult = table.getRow("enclosure1");

- Scan through a range of rows
  Scanner s = table.getScanner(new String[]{"animal:type"});
Hive

Original Slides by
Matei Zaharia
UC Berkeley RAD Lab
Hive

- Developed at Facebook
- Used for majority of Facebook jobs
- “Relational database” built on Hadoop
  - Maintains list of table schemas
  - SQL-like query language (HiveQL)
  - Can call Hadoop Streaming scripts from HiveQL
  - Supports table partitioning, clustering, complex data types, some optimizations
Creating a Hive Table

CREATE TABLE page_views(viewTime INT, userid BIGINT,
                        page_url STRING, referrer_url STRING,
                        ip STRING COMMENT 'User IP address')
COMMENT 'This is the page view table'
PARTITIONED BY(dt STRING, country STRING)
STORED AS SEQUENCEFILE;

- Partitioning breaks table into separate files for each (dt, country) pair
  Ex: /hive/page_view/dt=2008-06-08,country=USA
      /hive/page_view/dt=2008-06-08,country=CA
A Simple Query

• Find all page views coming from xyz.com on March 31st:

```sql
SELECT page_views.*
FROM page_views
WHERE page_views.date >= '2008-03-01'
AND page_views.date <= '2008-03-31'
AND page_views.referrer_url like '%xyz.com';
```

• Hive only reads partition 2008-03-01,* instead of scanning entire table
Aggregation and Joins

• Count users who visited each page by gender:

```
SELECT pv.page_url, u.gender, COUNT(DISTINCT u.id)
FROM page_views pv JOIN user u ON (pv.userid = u.id)
GROUP BY pv.page_url, u.gender
WHERE pv.date = '2008-03-03';
```

• Sample output:

<table>
<thead>
<tr>
<th>page_url</th>
<th>gender</th>
<th>count(userid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>home.php</td>
<td>MALE</td>
<td>12,141,412</td>
</tr>
<tr>
<td>home.php</td>
<td>FEMALE</td>
<td>15,431,579</td>
</tr>
<tr>
<td>photo.php</td>
<td>MALE</td>
<td>23,941,451</td>
</tr>
<tr>
<td>photo.php</td>
<td>FEMALE</td>
<td>21,231,314</td>
</tr>
</tbody>
</table>
SELECT TRANSFORM(page_views.userid,
                     page_views.date)
USING 'map_script.py'
AS dt, uid CLUSTER BY dt
FROM page_views;
Storm

Original Slides by
Nathan Marz
Twitter
Storm

• Developed by BackType which was acquired by Twitter
• Lots of tools for data (i.e. batch) processing
  – Hadoop, Pig, HBase, Hive, ...
• None of them are realtime systems which is becoming a real requirement for businesses
• Storm provides realtime computation
  – Scalable
  – Guarantees no data loss
  – Extremely robust and fault-tolerant
  – Programming language agnostic
Before Storm
Before Storm – Adding a worker

- Twitter firehose
- Hadoop
- Cassandra

Deploy

Reconfigure/Redeploy
Problems

- Scaling is painful
- Poor fault-tolerance
- Coding is tedious
What we want

• Guaranteed data processing
• Horizontal scalability
• Fault-tolerance
• No intermediate message brokers!
• Higher level abstraction than message passing
• “Just works”!!
Master node (similar to Hadoop JobTracker)

Used for cluster coordination

Run worker processes
Concepts

- Streams
- Spouts
- Bolts
- Topologies
Streams

Unbounded sequence of tuples
Spouts

Source of streams
Processes input streams and produces new streams:
Can implement functions such as filters, aggregation, join, etc
Network of spouts and bolts
Spouts and bolts execute as many tasks across the cluster.
Stream Grouping

When a tuple is emitted which task does it go to?
Stream Grouping

- **Shuffle grouping**: pick a random task
- **Fields grouping**: consistent hashing on a subset of tuple fields
- **All grouping**: send to all tasks
- **Global grouping**: pick task with lowest id